**Engr 304 – Software Lab 4 Answer Template**

**Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Section: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Part I:**

**Insert your code for calculating the SumArray here, using proper formatting.**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Name: Daniel Ackuaku \*/

/\* Course: Engineering 304L \*/

/\* Lab: Lab Number 4 \*/

/\* Date: 28th February 2019. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* The ".include nios\_macros.s" assembler directive includes NIOS macros for use in the program \*/

.include "nios\_macros.s"

/\* The word-aligned address of the reset vector, value is taken from the

cpu configuration in SOPC builder. \*/

/\*TEMPLATE: replace 0x00 with the appropriate address \*/

.equ RESET\_VECTOR, 0x00

/\* The ".text" assemlber directive indicates the beginning of the code section of the program \*/

.text

/\* The ".org RESET\_VECTOR" assembler directive places the main routine at the reset address \*/

.org RESET\_VECTOR

/\* The ".global \_start" assembler directive exports the "\_start" label as an external symbol \*/

.global \_start

/\* The "\_start" label identifies the program start location for the debugger \*/

\_start:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\* MAIN \*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Pseudo-code for this program is as follows \*/

/\* Initialize variables (sum), initialize registers, etc

Begin LOOP

Read value from array

If value=0, exit the loop, else

Add value to sum variable

Increment loop index (or pointer address)

return to "Begin LOOP"

\*/

/\*\*\*\* init \*\*\*\*/

MAIN\_PROG\_INIT:

movia gp, SumArray

movia sp, 0xffff /\*ffff/\* load the address of the array into the gp register \*/

movi r17, 50

mov r4, r0 /\* clear register 8 for the sum \*/

/\*\*\*\* run \*\*\*\*/

MAIN\_PROG:

beq r17, r4, MAIN\_PROG\_END

call SumCalc

addi r4, r4, 1

stw r2, 0(gp)

addi gp, gp, 4

br MAIN\_PROG

/\*\*\*\* destroy \*\*\*\*/

MAIN\_PROG\_END:

br MAIN\_PROG\_END /\* infinite loop to keep program from going into the weeds \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\* DATA \*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* The ".data" directive identifies the section of the program that defines global variables \*/

.data

/\* The "SumArray" label has an address equal to the address of the first element in the

array of words immediately following the label, this program uses it as arbitrary

zero-terminated example content to sum in the main program. \*/

SumArray:

.skip 50\*4

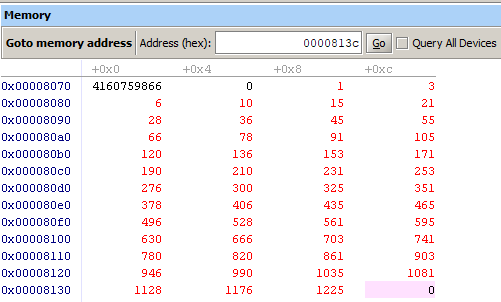
/\* The ".end" assembler directive indicates the end of the program and

all following lines are discarded \*/

.end

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Insert your screenshot showing your array in memory in decimal format**



**Part II:**

**Provide your equations for stack space and for code/global space here, providing explanations for how you arrived at the equations.**

1. 3 items are pushed on the stack each time SumCalc is called.
2. Each new number takes up 12 bytes in the stack. Since a word is 4 bytes the stack entry for a single top level SumCalc(n) = 3\*n
3. Since each new entry takes up 4 bytes so in terms of n the stack would need to be 12\*n
4. The main program uses the addresses 8000 to 8070 which corresponds to 112 bytes. Whereas the global array uses up addresses 0x8074 to 0x813c = 200 bytes which is accordance with the size of memory allocated for the address at the beginning of the program 50\*4 = 200 bytes.
5. Total memory = 112 + (12+4) \*n
6. 0x00ffff – 0x00800 = 32767 (bytes between the stack and main memory). Therefore the number of n terms required for the stack to run into the global variables is approximately equal to to be 32767 / Total memory.

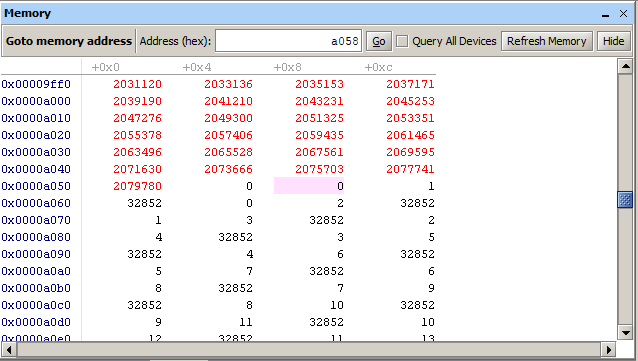
**Provide your maximum safe value of “n” here, showing your calculations**

Safe n = (32766 – 112) /16 = 2040 terms.

**Fill in this table with the range of addresses used by the parts of memory when using the maximum safe value of n.**

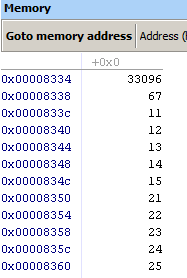
|  |  |
| --- | --- |
| **Memory Type** | **Address Range (in hex)** |
| Instructions (.text) | 0x008000 – 0x008070 |
| Global Variables (.data) | 0x008074 – 0xa054 |
| Stack | 0xa050c – 0x00ffff |

**Provide your screenshot of your array and your stack almost colliding in memory**



**Part III:**

**Provide your screenshot of the array in memory (single element per row, in decimal)**



**Fill in this table with “X”s in the appropriate columns.**

|  |  |  |
| --- | --- | --- |
| **Language** | **Uses Row-Major** | **Uses Column-Major** |
| C | X |  |
| C++ | X |  |
| Fortran |  | X |
| Python | Neither | Neither |
| Mathematica | Neither | Neither |
| Pascal | X |  |
| MatLab |  | X |
| R (statistics) |  | X |

Information form: <https://en.wikipedia.org/wiki/Row-_and_column-major_order>

**Extra Credit:**

**Show your calculations of the number of instructions executed plus the estimate of the program execution time and show the comparison of that execution time with an actual run.**